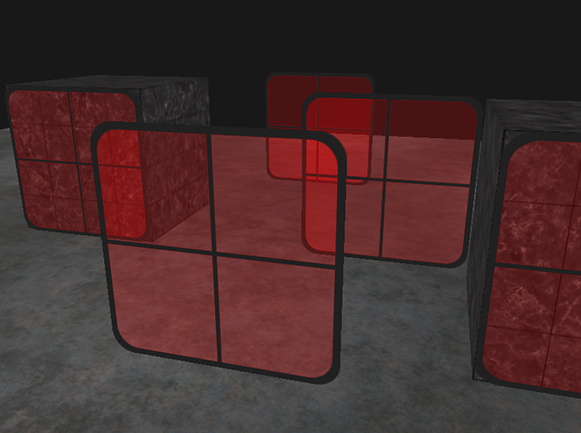
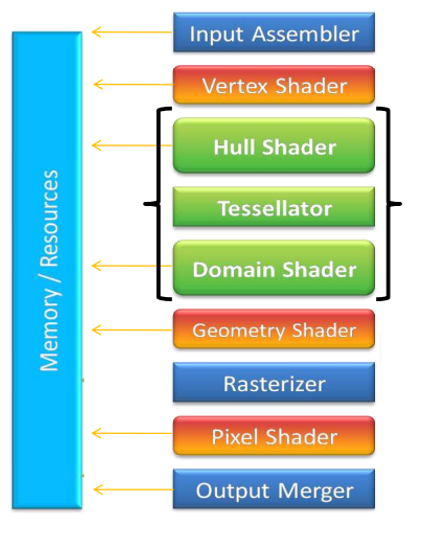
Blending and Alpha Clipping

QUIZ!

Consider the following image:



1. A pixel without transparency has three components, with transparency it has four. Name these four components
2. What effect would the following values have on transparency? 0, 0.3, 0.9?
3. There are three glass panels in the scene. In the simplest terms, how do the pixels for each panel look transparent? So, what is done to the colour of the pixel?
4. For the above image, what consideration does the developer have to make regarding the draw order of the scene?
5. What might happen if the draw order was ignored?
6. As well as the draw order, what normally important buffer does the developer have to modify the behaviour of, and why?
7. At what stage in the directX pipeline does blending occur? 
8. What is the difference between blending and clipping?
9. Can you think of an example of when clipping might be useful?
10. What is the difference between a blending operator and a blending factor? Can you describe an example of how these might be used to produce thick dark smoke in a game? So – where the smoke is not very thick it would appear light, where it is very thick it would appear dark?

**In this tutorial we will be implementing transparency and creating a texture with a number of pixels clipped, this will allow us to increase the realism of our scenes and add a number of extra elements.**

# Blending Walkthrough

## The Blending Equation

In Direct3D, to create the illusion of transparency, we use an equation which will take the pixels behind the transparent primitive on the render target and blend their colours with the current transparent primitives pixels. The blending equation has a number of parts:

(FC) - Final Color

(SP) - Source Pixel

(DP) - Destination Pixel

(SBF) - Source Blend Factor

(DBF) - Destination Blend Factor

(FA) - Final Alpha

(SA) - Source Alpha

(DA) - Destination Alpha

(+) - Binary Operator described below

Direct3d uses two different blending equations, one for colour and one for alpha. There are two different equations for colour and alpha because we want to process them both differently, using different operators and different blend factors. The two equations are as follows:

(FC) = (SP)(SBF) (+) (DP)(DPF)

(FA) = (SA)(SBF) (+) (DA)(DBF)

There are a number of binary (+) operators that can be used such as Add, Min, Max, etc. They are defined in the D3D11\_BLEND\_OP enum.

We use blend factors to achieve different effects when blending. We can set blend factors with the D3D11\_BLEND enumerated type. They are defined in the D3D11\_BLEND enum.

More information on the different enums and options available can be found [here](http://msdn.microsoft.com/en-gb/library/windows/desktop/bb205072(v=vs.85).aspx).

We create a blend description to tell direct3d how we will be blending our pixels. To create a blend description, we need to fill out a D3D11\_BLEND\_DESC structure (already defined in Direct3D):

typedef struct D3D11\_BLEND\_DESC {

BOOL AlphaToCoverageEnable;

BOOL IndependentBlendEnable;

D3D11\_RENDER\_TARGET\_BLEND\_DESC RenderTarget[8];

} D3D11\_BLEND\_DESC;

This blend description contains an array of different D3D11\_RENDER\_TARGET\_BLEND\_DESC structures which define the blending equation for each render target (we can have 8 in total).

The D3D11\_RENDER\_TARGET\_BLEND\_DESC is defined like this (again, already defined in Direct3D):

typedef struct D3D11\_RENDER\_TARGET\_BLEND\_DESC {

BOOL BlendEnable;

D3D11\_BLEND SrcBlend;

D3D11\_BLEND DestBlend;

D3D11\_BLEND\_OP BlendOp;

D3D11\_BLEND SrcBlendAlpha;

D3D11\_BLEND DestBlendAlpha;

D3D11\_BLEND\_OP BlendOpAlpha;

UINT8 RenderTargetWriteMask;

} D3D11\_RENDER\_TARGET\_BLEND\_DESC;

**BlendEnable -**

Specify true to enable blending for this render target.

**SrcBlend -**

This is our source blend factor (SBF). We can set this to any of the enumerated D3D11\_BLEND types.

**DestBlend -**

This is our destination blend factor (DBF). We can set this to any of the enumerated D3D11\_BLEND types.

**BlendOp -**

Here is where we specify the blending operation to use which we discussed earlier. Set this to any of the D3D11\_BLEND\_OP enumerated types.

**SrcBlendAlpha -**

This is our source blend factor for the alpha channel (SBF). We can set this to any of the enumerated D3D11\_BLEND types.

**DestBlendAlpha -**

This is our destination blend factor for the alpha channel (SBF). We can set this to any of the enumerated D3D11\_BLEND types.

**BlendOpAlpha -**

Here we specify the blending operation to use for the alpha channels. Set this to any of the D3D10\_BLEND\_OP enumerated types.

**RenderTargetWriteMask -**

This is where we specify which channel to blend. We are able to choose R, G, B, A, All of them, or a combination of them.

# Blending Code – Transparency

Add the following to your application header file:

ID3D11BlendState\* Transparency;

Don't forget to release this in clean up!

## The Blending Equation

Now, at the bottom of our init device function, we will define our blending equation, then create it. We can create the blending State by calling the ID3D11Device::CreateBlendState(), where the first parameter is a pointer to our blending description, and the second is a pointer to an ID3D11BlendState interface.

The code for a blend state that uses an external blend factor is:

D3D11\_BLEND\_DESC blendDesc;

ZeroMemory( &blendDesc, sizeof(blendDesc) );

D3D11\_RENDER\_TARGET\_BLEND\_DESC rtbd;

ZeroMemory( &rtbd, sizeof(rtbd) );

rtbd.BlendEnable = true;

rtbd.SrcBlend = D3D11\_BLEND\_SRC\_COLOR;

rtbd.DestBlend = D3D11\_BLEND\_BLEND\_FACTOR;

rtbd.BlendOp = D3D11\_BLEND\_OP\_ADD;

rtbd.SrcBlendAlpha = D3D11\_BLEND\_ONE;

rtbd.DestBlendAlpha = D3D11\_BLEND\_ZERO;

rtbd.BlendOpAlpha = D3D11\_BLEND\_OP\_ADD;

rtbd.RenderTargetWriteMask = D3D10\_COLOR\_WRITE\_ENABLE\_ALL;

blendDesc.AlphaToCoverageEnable = false;

blendDesc.RenderTarget[0] = rtbd;

d3d11Device->CreateBlendState(&blendDesc, &Transparency);

Finally, in our draw function we set the correct blend state for the objects we are drawing:

// "fine-tune" the blending equation

float blendFactor[] = {0.75f, 0.75f, 0.75f, 1.0f};

// Set the default blend state (no blending) for opaque objects

d3d11DevCon->OMSetBlendState(0, 0, 0xffffffff);

// Render opaque objects //

// Set the blend state for transparent objects

d3d11DevCon->OMSetBlendState(Transparency, blendFactor, 0xffffffff);

These new lines of code set our blending equation to be used. Remember, blending works by taking what’s already on the render target, and blending the current objects colours with what’s already there. So, this being the case, we need to make sure we render the opaque objects first, so that the transparent objects can blend with the opaque objects.

The first line is a blend factor. We will use this blend factor to determine the transparency of our boxes. This blend factor says that the red, green, and blue colours on the current primitive will be 25% transparent. This could also mean that the primitive will be 75% transparent, or even nothing if you were to change the blending description.

The next line will disable the blend state, so that we can draw our opaque objects. We can turn off blending by setting the first two parameters here to NULL, and the third to 0xffffffff. Then we turn blending on, by calling the method.

The ID3D11DeviceContext::OMSetBlendState() method binds the blend state of our choice to the OM stage of the pipeline, where it will use the blending equation to implement the transparent effect (blending is not always transparent, you can do many things with blending). The first parameter is the ID3D11BlendState object, the second is an array of 4 floats (RGBA), and the last has to do with sample coverage, 0xffffffff is the default.

Have a play around with some of the different settings available to get an idea of the different effects that are available when using blending.

# Alpha Clipping

We can completely discard a pixel from further processing by using the clip() function in the pixel shader. Usually we use the textures alpha component to decide whether to discard the pixel or not.

An example of the clip function in use:

clip(alpha - 0.25f);

This will clip the pixel if the alpha value is 0.25 or less. How the clip function works, is if the value is less than zero then it will stop the pixel shader completely from further processing, and no data will be written to the render target OR the depth/stencil buffer.

So, to enable this we simply need to add the able line of code to our pixel shader, and to make sure that the alpha channel of our textures have the correct clipping value.

If you wanted to have an object like a cage be displayed, then you would need to disable backface culling before rendering the object. This can be done by creating a new rasterizer description with the cullmode set to D3D11\_CULL\_NONE.

# Exercise

1. Add in a transparent object into your scene, this could be water, glass in windows, or just an example object to test the implementation. Check that your scene is rendering correctly and that the transparency is correct. Many free textures can be found online which will help you better represent your in game objects.
2. Add a new vertical plane into your game and texture it with the Pine Tree texture that can be found on blackboard. This texture has correct alpha values to enable clipping. Observe the result when not using the clip command in the pixel shader.
3. Now add the clip command to the pixel shader, you should see that the areas around the tree are now removed.
4. Often we want to have a plane that always faces the camera. These are usually called Billboards. Information on creating a billboard world matrix can be found in the previous lecture called ‘Environmental Effects’. Use the built in DirectX functions (E.g. XMVector3Normalise, XMVector3Cross, etc) to create a billboard world matrix for your tree. If this is implemented correctly then you should see that the tree always faces the camera.

More information on billboards can be found [here](http://nehe.gamedev.net/article/billboarding_how_to/18011/). Note that this article uses the right handed coordinate system, whereas we use the left handed coordinate system. You will need to change some of the formulae accordingly.

1. Now add a number of billboarded trees into your world, you could randomise their positions, hardcode values into your application, or load them from an external file.

**Further Reading / Research Questions**

Consider the following problem (you do not need to implement this!).

1. You have been asked to develop a shader and produce textures which do the following: There will be a glass window with a wooden frame. The glass window will be transparent, but will also blur the contents of a scene behind it. In the middle of the glass window, a circular region is cut out (a hole). How would you achieve this effect with shaders, textures, and just two triangles (you can assume the surface is flat, and the cut out part of the window is still solid as far as the game physics is concerned).
2. There are many resources in books and online which discuss shadow rendering using stencil buffers. Take the following resource and write a short step-by-step process of how stencil buffer shadow volumes are implemented:
   1. <https://en.wikipedia.org/wiki/Shadow_volume>
   2. What are some limitations with stencil buffer shadow volume rendering? This will depend on the implementation – you will need to do further reading on this.